Algorithmics (6EAP)

Regular Expressions and Automata

Jaak Vilo

2020 fall

Contents

- Regular expressions and regular languages
- Automata
  - Deterministic finite automata **DFA**
  - Nondeterministic finite automata **NFA**
- Regular expressions to NFA
- NFA to DFA

Links

- **Navarro and Raffinot** Flexible Pattern Matching in Strings. (Cambridge University Press, 2002). ch. 5: Regular Expression Matching (pp. 99--143)
- Regular expression search using a DFA (relative difficulty: medium-hard) [ASU1986, pp. 92-105, 113-146], [NaRa2002, ch. 5], [Orponen1994, ch. 2]
- A. Aho: Algorithms for finding patterns in strings. In Handbook of Theoretical Computer Science, Vol. A, Elsevier, 1990, 255-300. (see library)
- Teoreetiline Informaatika (Jaan Penjam, TTÜ), Peatükk 5.
- Regulaarsest avaldisest mittedeterminееritud automaadi moodustamine (Meelis Roos) (kohalik)
- Google – Query
- http://en.wikipedia.org/wiki/Regular_expression

- GNU grep manual (grep = Global Search for Regular Expression and Print) http://www.hmug.org/man/1/grep.php
- FSA Utilities toolboxFSA Utilities toolbox: a collection of utilities to manipulate regular expressions, finite-state automata and finite-state transducers. (Gertjan van Noord)
- Finnish-language course Models for Programming and Computing - essential regular expressions and automata theory...
- http://www.regular-expressions.info/

Regular expression

- **Definition:** A *regular expression* RE is a string on the set of symbols $\Sigma \cup \{ \varepsilon , | , \cdot , * , (,) \}$, which is recursively defined as follows. RE is
  - an empty character ε ,
  - a character $\alpha \in \Sigma$,
  - (RE_1),
  - ($RE_1 \cdot RE_2$),
  - ($RE_1 | RE_2$), and
  - (RE_1*),
  - where RE_1 and RE_2 are regular expressions

Example

(((A · T) | (G · A)) · (((A · G) | ((A · A) · A))*))

- we can simplify

  (AT|GA)((AG|AAA)*)

- Often also this is used:

  RE+ = RE · RE*

Why?

- Regular expression defines a language
- A set of words from Σ^*
- A convenient short-hand
- (AT|GA)((AG|AAA)*) => AT , ATAG , GAAAA, GAAGAAAAAA, …
- Infinite set

Language represented by RE

Definition: A *language represented by a regular expression* RE us a set of strings over Σ, which is defined recursively on the structure of RE as follows:

- if RE is ε, then L(RE)={ε}, the empty string
- if RE is $\alpha \in \Sigma$, then L(RE)={α}, a single string of one character
- if RE is of the form (RE_1), then $L(RE)=L(RE_1)$
- if RE is of the form $(RE_1 \cdot RE_2)$, then $L(RE)=L(RE_1) \cdot L(RE_2)$, where $w=w_1w_2$ is in L(RE) if $w_1 \in L(RE_1)$ and $w_2 \in L(RE_2)$. (We call · the concatenation operator)
- if RE is of the form $(RE_1 \mid RE_2)$, then $L(RE)=L(RE_1)\ L(RE_2)$, the union of two languages. (We call | the union operator)
- if RE is of the form (RE_1^*), then $L(RE) = L(RE)^* = \bigcup_{i \geq 0} L(RE_1)^i$, where $L^0 = \{ \varepsilon \}$ and $L^i = L \cdot L^{i-1}$. (We call * the star operator)

Regular expression	Language L(RE)	Comment
ε	$\{\varepsilon\}$	Empty string
$\alpha \in \Sigma$	$\{\alpha\}$	Single character
(RE_1)	$L(RE_1)$	Parenthesis
$(RE_1 \cdot RE_2)$	$(RE_1 \cdot RE_2)$ $w=w_1w_2$ is in L(RE) if $w_1 \in L(RE_1)$ and $w_2 \in L(RE_2)$	Concatenation
$(RE_1 \mid RE_2)$	$L(RE_1) \cup L(RE_2)$	Union
(RE_1^*)	$L(RE)^* = \cup_{i \geq 0} L(RE_1)^i$ $L^0 = \{ \varepsilon \}$ and $L^i = L \cdot L^{i-1}$.	The star operator (Kleene star)
(RE_1+)	$(RE_1) \cdot (RE_1^*)$	Kleene plus

- L((AT|GA)((AG|AAA)*)) = { AT, GA , ATAG , GAAG, ATAAA, GAAAA, ATAGAG, ATAGAAA, ATAAAAG, ... }

- Σ* denotes all strings over alphabet Σ
- The *size* of a regular expression RE is the number of characters of Σ in it.
- Many complexities are based on this measure.

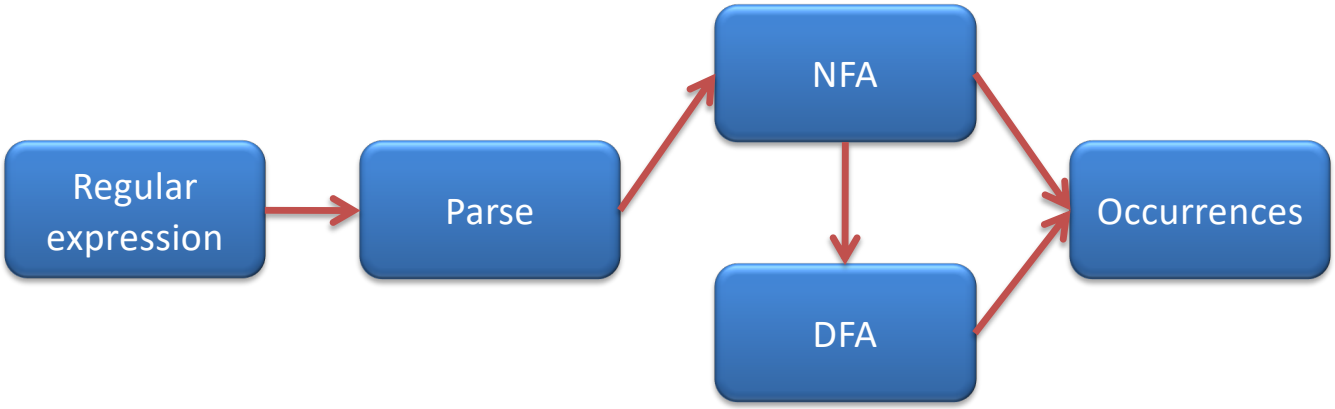
A different example definition

- Just as finite automata are used to *recognize* patterns of strings, regular expressions are used to *generate* patterns of strings. A regular expression is an algebraic formula whose value is a pattern consisting of a set of strings, called the language of the expression.
- Operands in a regular expression can be:
- *characters* from the alphabet over which the regular expression is defined.
- *variables* whose values are any pattern defined by a regular expression.
- *epsilon* which denotes the empty string containing no characters.
- *null* which denotes the empty set of strings.
- Operators used in regular expressions include:
- * Concatenation: If R1 and R2 are regular expressions, then R1R2 (also written as R1.R2) is also a regular expression.
- L(R1R2) = L(R1) concatenated with L(R2).
- * Union: If R1 and R2 are regular expressions, then R1 | R2 (also written as R1 U R2 or R1 + R2) is also a regular expression.
- L(R1|R2) = L(R1) U L(R2).
- * Kleene closure: If R1 is a regular expression, then R1* (the Kleene closure of R1) is also a regular expression.
- L(R1*) = epsilon U L(R1) U L(R1R1) U L(R1R1R1) U ...
- Closure has the highest precedence, followed by concatenation, followed by union.

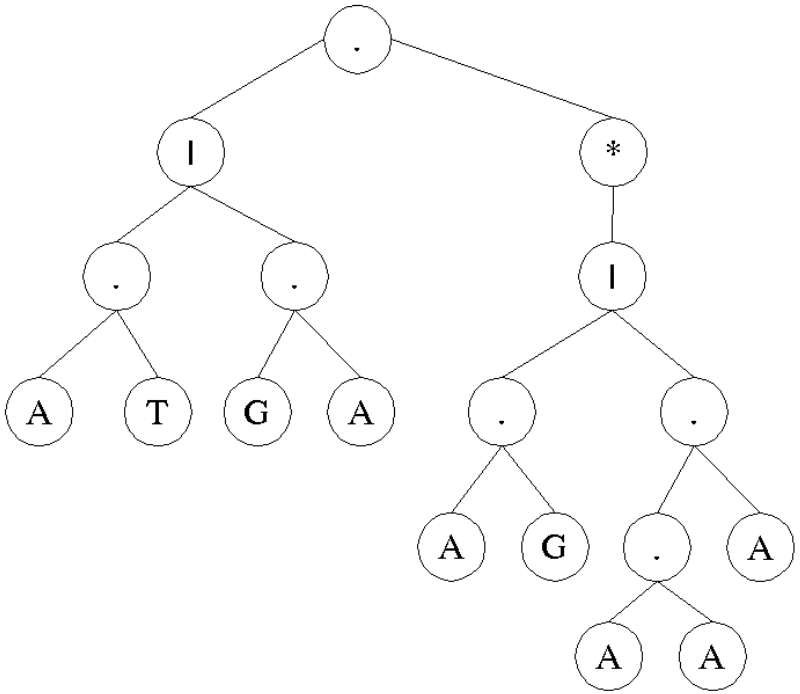
Regexp Matching

- The problem of searching regular expression RE in a text T is to find all the factors of T that belong to the language L(RE).
- Parsing
- Thompsons NFA construction (1968)
  Glushkov NFA construction (1961)
- Search with the NFA
- Determinization
- Search with the DFA
- Or, search by extracting set of strings, multi-pattern matching, and verification of real occurrences.

Matching of RE-s

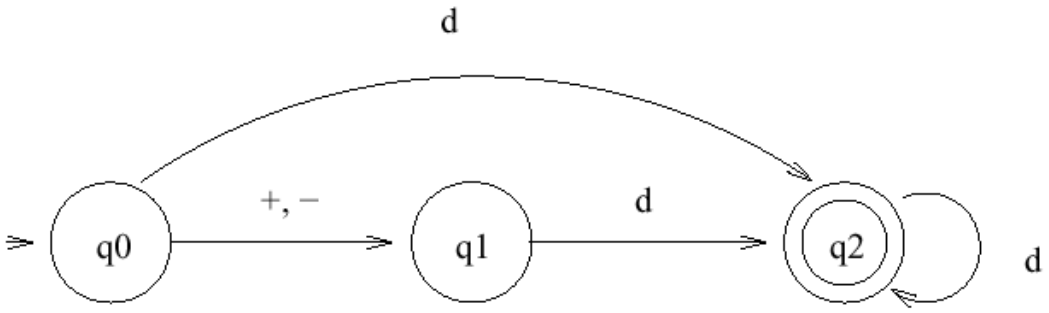


Parse tree

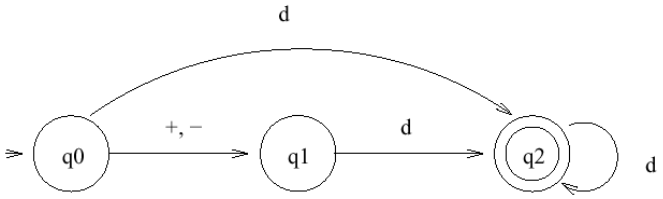


(A T | G A)((A G | A A A)*)

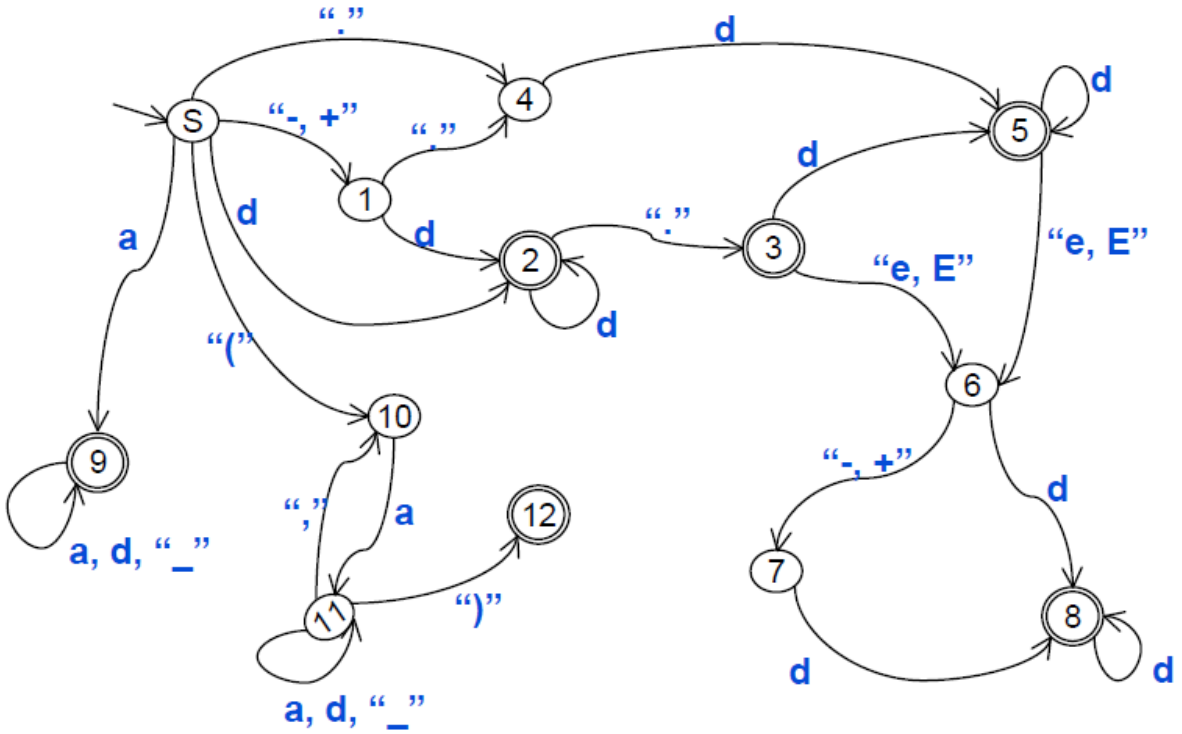
Q: what is the language?



```
Function IsDigit(c) {  if ( c ∈ { 0,1,...,9 } ) return 1 else return 0 }
int q=0 ; // current state
int sign=1 ; // sign of the value int
val=0 ; // value of the number
while( ( c=getc() ) != EOF )
        switch ( q ) {
        case 0 :   if c ∈ { '+', '-' } q = 1
                   if ( c == '-' ) sign= -1
                   elsif          IsDigit(c) val = c - '0' // numeric value of c
                                  q = 2
                   else q = 99
                   break ;
        case 1 :   if IsDigit(c)
                                  val = c - '0'
                                  q = 2
                   else q = 99
                   break ;
        case 2 :
                   if IsDigit(c)
                                  val = 10*val + ( c - '0')
                                  q = 2
                   else q = 99
                    break ;
        case 99 : break ;
}
if( q == 2 )
 then print 'The value of the number is ' , sign*val
else print ‘Does not match the automaton for signed integers’
```



Lõplik automaat (näide)



Lõppolekud:

- 2: INTEGER
- 5: REAL
- 8: Scientific
- 9: Identifier
- 12: Identifier List

Jaan Penjam: http://www.cs.ioc.ee/~jaan/teorinf/konspekt/loeng2.pdf

Deterministic finite automaton DFA

Definition DFA is a quintuple $M=(Q, \Sigma, \delta, q_0, F)$, where

- Q is the finite set of states of an automaton
- Σ is the input alphabet
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is a set of accepting final states

Usage:

- Transition step - $(q, aw) \vdash (q' , w)$ if $q' = \delta(q,a)$, $w \in \Sigma^*$
- Accepted language: $L(M) = \{ w \mid (q_0 , w) \vdash^* (q, \varepsilon), q \in F \}$

Non-deterministic finite automaton NFA

Definition NFA is a quintuple $M=(Q, \Sigma, \delta, q_0, F)$, where

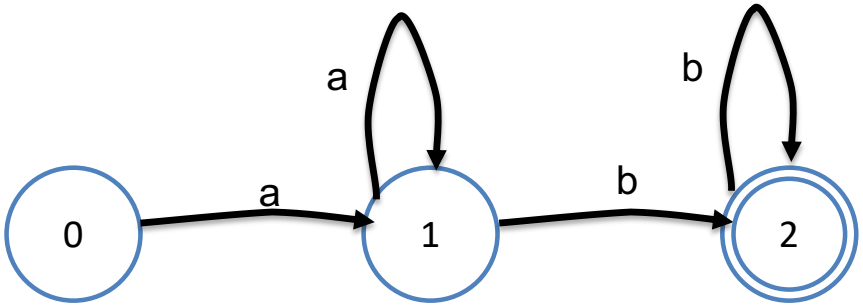
- Q is the finite set of states of an automaton
- Σ is the input alphabet
- $\delta : Q \times \Sigma \rightarrow \mathbf{P(Q)}$ is the transition function (a set)
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is a set of accepting final states

Usage:

- Transition step - $(q, aw) \vdash (q' , w)$ if $\mathbf{q' \in \delta(q,a)}$, $\mathbf{a \in \Sigma \cup \{ \varepsilon \}}$, $w \in \Sigma^*$
- Accepted language: $L(M) = \{ w \mid (q_0 , w) \vdash^* (q, \varepsilon), q \in F \}$

DFA

a+ b+



$Q = \{ S_0 , S_1, S_2 \}$
$\Sigma = \{ a , b \}$
δ :

State	Char	State
0	a	1
1	a	1
1	b	2
2	b	2

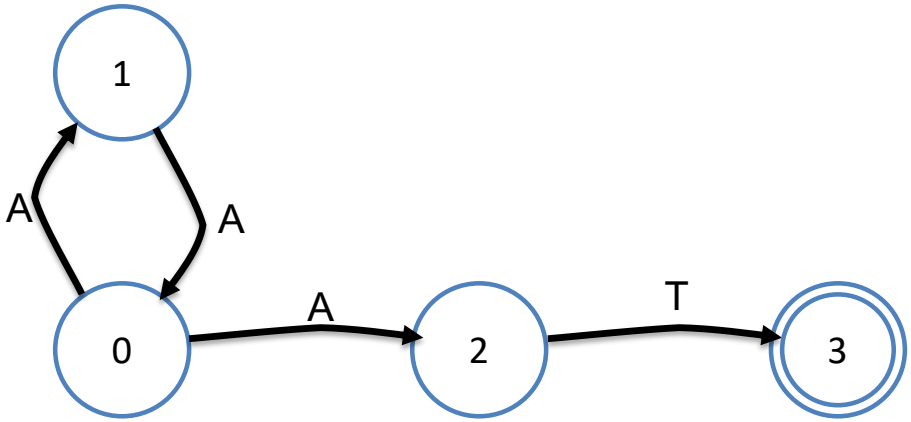
$q_0 \in S_0$
$F = \{ S_2 \}$

$S_0 \rightarrow a\ S_1$
$S_1 \rightarrow a\ S_1$
$S_1 \rightarrow b\ S_2$
$S_2 \rightarrow b\ S_2$

$S_0 \rightarrow a\ S_1 \rightarrow a\ S_1 \rightarrow a\ S_1 \rightarrow a\ S_1 \rightarrow b\ S_2 \rightarrow b\ S_2$

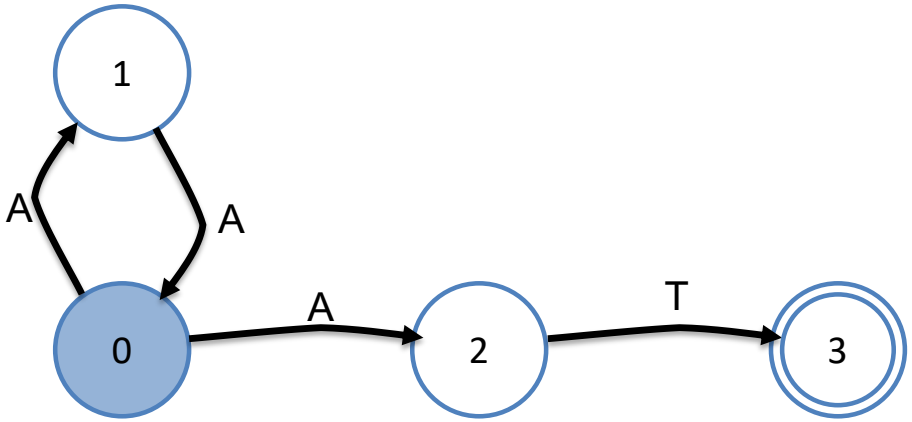
a a a a b b

(AA)*AT



AAAAAT

- (AA)*AT



AAAAAT

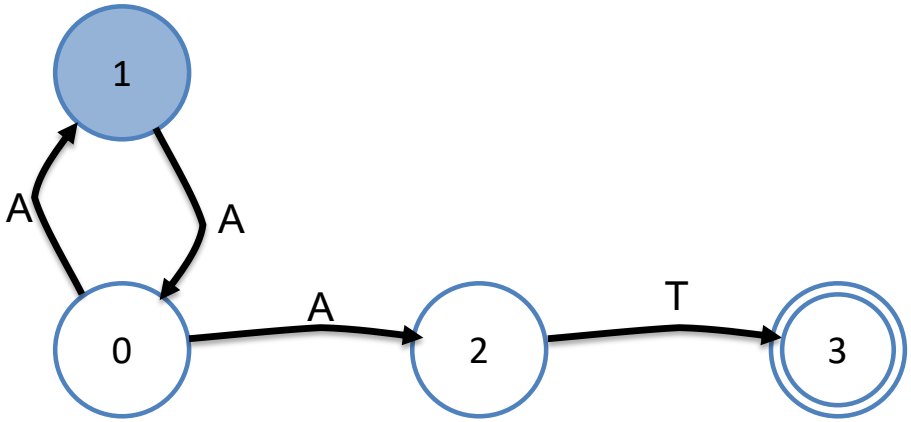
- (AA)*AT

$S_0 \rightarrow a\ S_1$

$S_1 \rightarrow a\ S_0$

$S_0 \rightarrow a\ S_2$

$S_2 \rightarrow t\ S_3$



A AAAAT

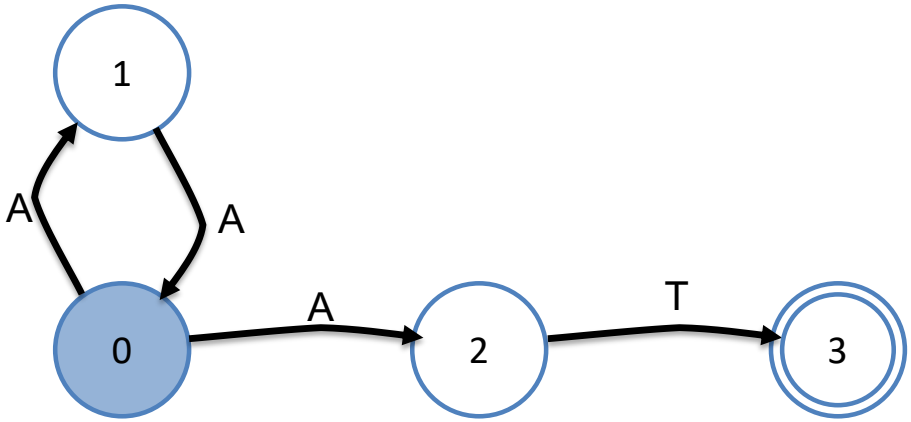
- (AA)*AT

$S_0 \rightarrow a\ S_1$

$S_1 \rightarrow a\ S_0$

$S_0 \rightarrow a\ S_2$

$S_2 \rightarrow t\ S_3$



A AAA AT

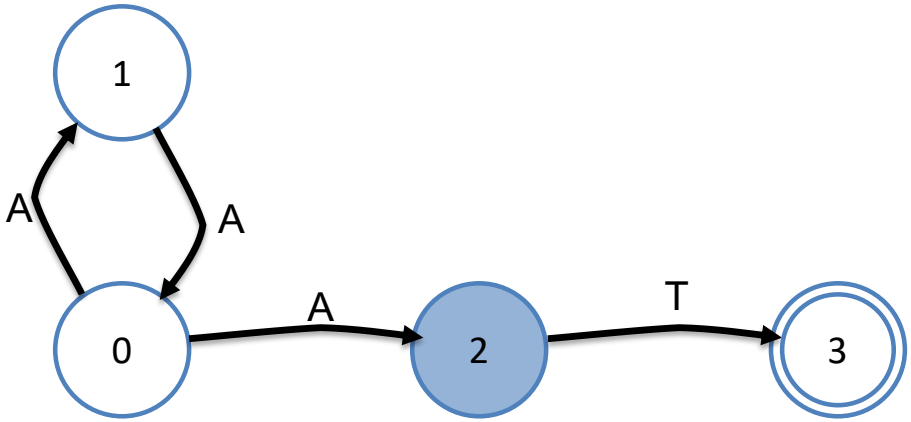
- (AA)*AT

$S_0 \rightarrow a\ S_1$

$S_1 \rightarrow a\ S_0$

$S_0 \rightarrow a\ S_2$

$S_2 \rightarrow t\ S_3$



A AAA A T

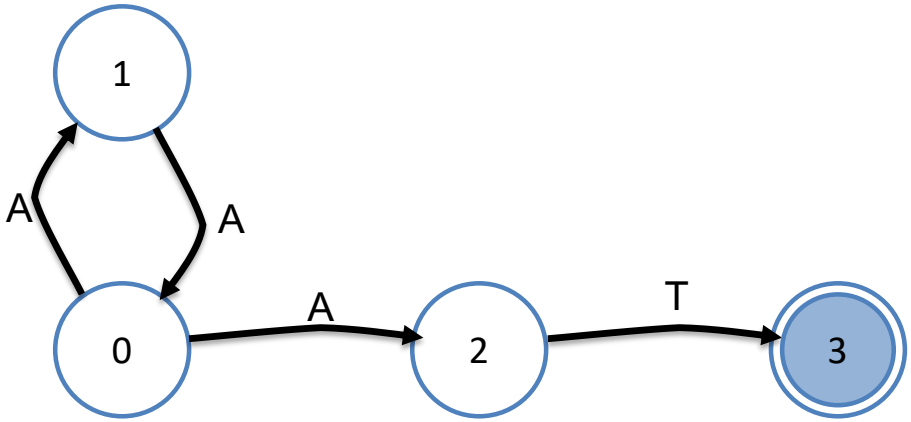
- (AA)*AT

$S_0 \rightarrow a\ S_1$

$S_1 \rightarrow a\ S_0$

$S_0 \rightarrow a\ S_2$

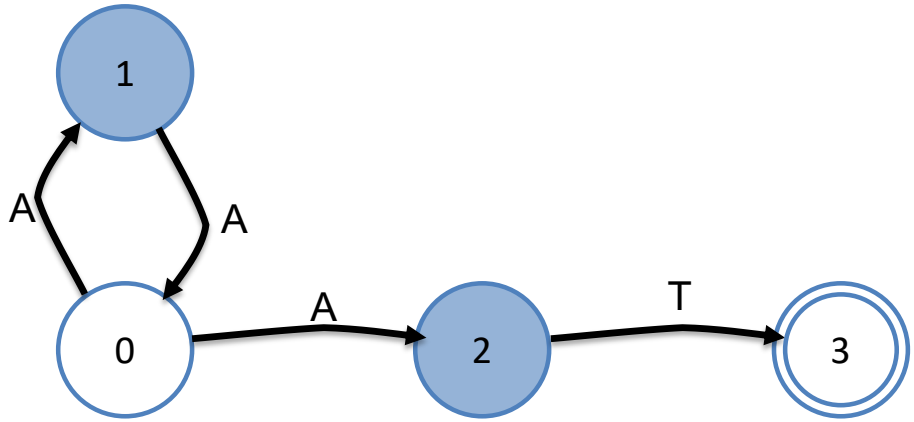
$S_2 \rightarrow t\ S_3$



A AAA AT

NFA – simultaneously in all reachable states

- (AA)*AT



$S_0 \rightarrow a\ S_1$

$S_1 \rightarrow a\ S_0$

$S_0 \rightarrow a\ S_2$

$S_2 \rightarrow t\ S_3$

A AAA A T

State	Char	State
0	a	1 , 2
1	a	0
2	b	2

Regexp -> NFA / DFA

- **Construction of an automaton from the regular expression**
- Regular expressions are mathematical and human-readable descriptions of the language
- Automata represent computational mechanisms to evaluate the language
- One needs to be able to parse the regular expression and to construct an automaton for matching it.

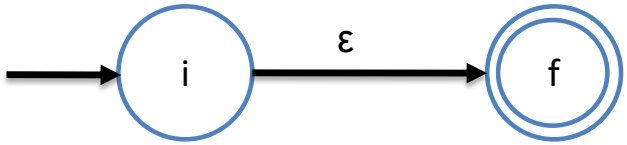
- See **Navarro and Raffinot** Flexible Pattern Matching in Strings. (Cambridge University Press, 2002). pp. 103.
- Automaadi konstruktsioon: Regulaarsest avaldisest mittedeterminеeritud automaadi moodustamine (Meelis Roos) (kohalik)
- Tsitaat:
- Kasutame Thompsoni konstruktsiooni. Selle konstruktsiooni idee on seada igale regulaaravaldises esineda võivale operatsioonile vastavusse primitiivne automaat. Kogu avaldisele vastab primitiivsete automaatide lihtne kompositsioon. Korduvatele alamavaldistele seatakse vastavusse mitu vastavalt mitu primitiivset automaati, mingit optimiseerimist siin ei toimu.
- Nii saadud lõplik automaat pole determineeritud, kuna me kasutame juba primitiivsetes automaatides mittedeterminеeritust. Lõpliku automaadi võib hiljem muidugi eraldi determineerida.

Thompson construction

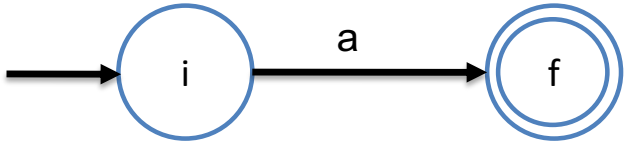
- Primitive automata
- Composition
- No optimality, no compression, etc.

Thompson construction: 2 primitive automata

- Symbol ε :

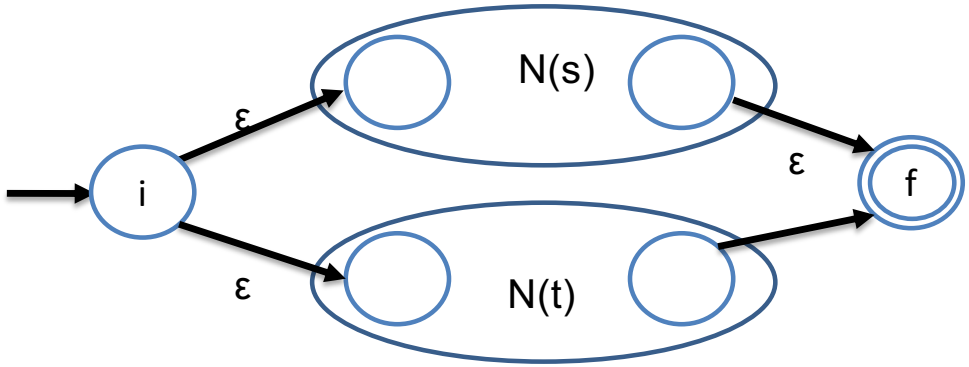


- Terminal symbol **a** :

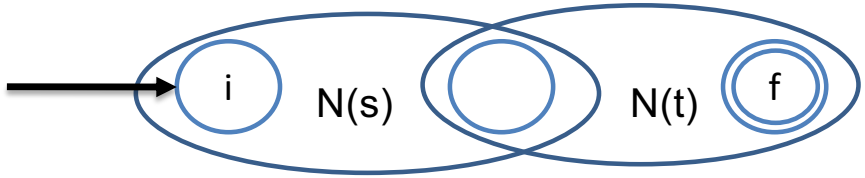


Union and Concatenation

- **s|t**

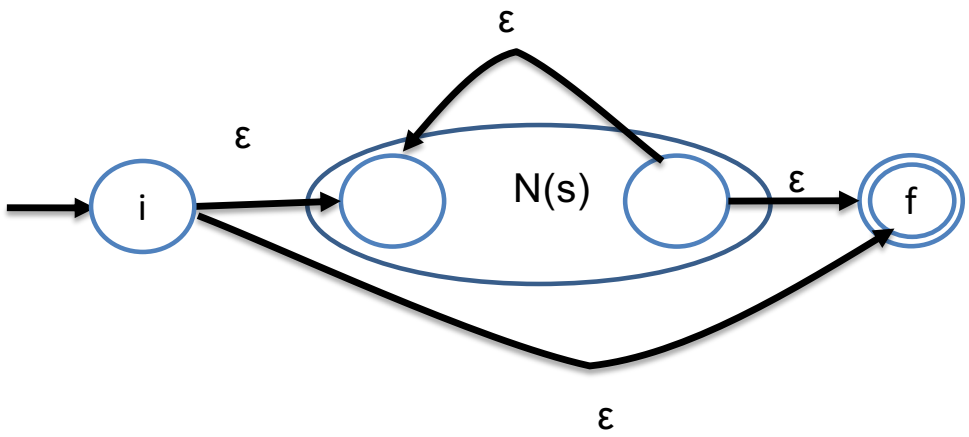


- **st**



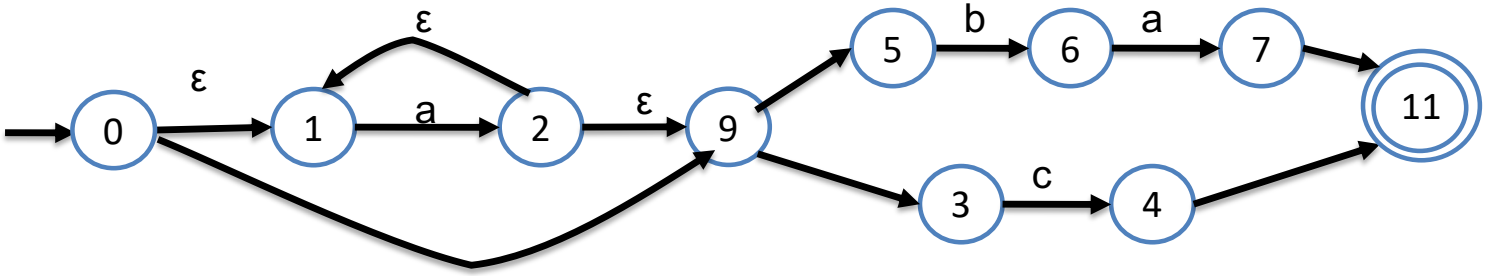
Closure

- **S***



Example

- **a*(ba|c)**



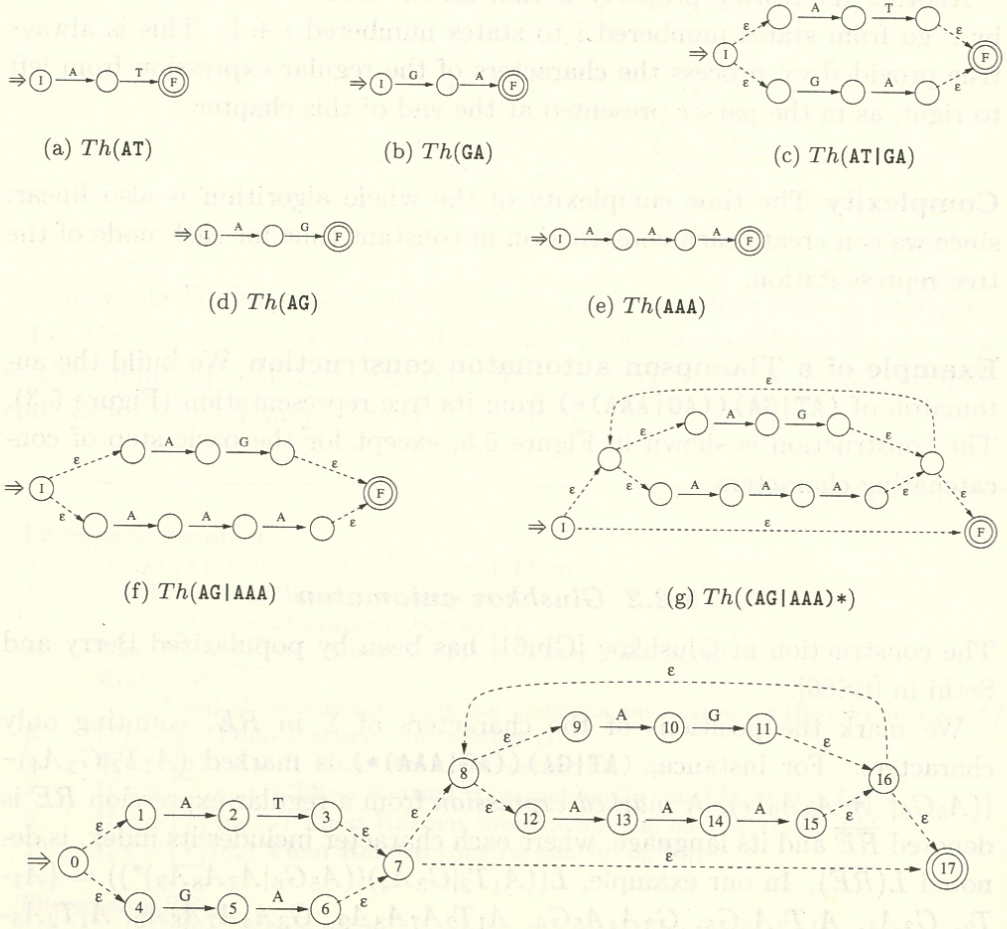


Fig. 5.5. Thompson automaton construction for the regular expression (AA|AT)((AG|AAA)*).

- Produces up to 2m states, but it has interesting properties, such as ensuring a linear number of edges, constant indegree and outdegree, etc.

Simulation of an NFA
Input: NFA $M_A=(Q, \Sigma, \delta, q_0, F)$, Text S = s[1..n]
Output: States after each character read Q0, Q1, ... Qn
NB: $S \in L(M_A)$ only if $F \subseteq Qn$.

Initially queue and sets Qi are empty

1 or more?

```
1. for i = 0 to n do                    // for each symbol of text
2.     mark all q ∈ Q unreached
3.     if  ( i == 0 )
4.     then                              // Initialise start state
5.             Q0 = q0;  queue = q0;  mark q0 as reached
6.     else
7.             foreach q ∈ Q(i-1)       // Main transitions on s[i]
8.                 foreach p ∈ δ(q, s[i])    // All transitions on s[i]
9.                     if p not yeat reached
10.                        Qi = Qi ∪ p
11.                        push( queue, p )
12.                        mark p as reached
13.    while queue ≠ ∅
14.        q = take( queue ) // Follow up on all ε - transitions
15.        foreach p ∈ δ(q, ε )   // All ε - transitions
16.                if p not yet reached
17.                    Qi = Qi ∪ p
18.                    push( queue, p )
19.                    mark p as reached
```

Epsilons?

- **Theorem** Time complexity of the NFA simulation is $O(|| M_A || \cdot n)$ where $||M_A||$ is the total number of states and transitions of M_A, $||M_A|| \leq 6 |A|$.
- **Proof** - During one step all states are manipulated only once, since all states are marked reached. There is at most n steps. The size of the automaton is at most $6 |A|$ where $|A|$ is the length of the regular expression.
-

Glushkov construction

$$(A_1 \; T_2 \mid G_3 \; A_4)((A_5 \; G_6 \mid A_7 \; A_8 \; A_9)^*)$$

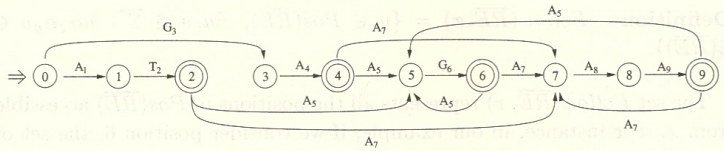


Fig. 5.6. Marked Glushkov automaton built on the marked regular expression $(A_1 T_2|G_3 A_4)((A_5 G_6|A_7 A_8 A_9)*)$. The state 0 is initial. Double-circled states are final.

To obtain the Glushkov automaton of the original RE, we simply erase the position indices in the marked automaton. At this step, the automaton usually becomes nondeterministic. The new automaton recognizes the language $L(RE)$. The Glushkov automaton of our example `(AT|GA)((AG|AAA)*)` is shown in Figure 5.7.

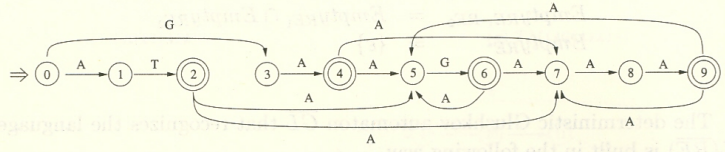


Fig. 5.7. Glushkov automaton built on the regular expression `(AT|GA)((AG|AAA)*)`. The state 0 is initial. Double-circled states are final. The automaton is derived from the marked automaton by simply erasing the position indices.

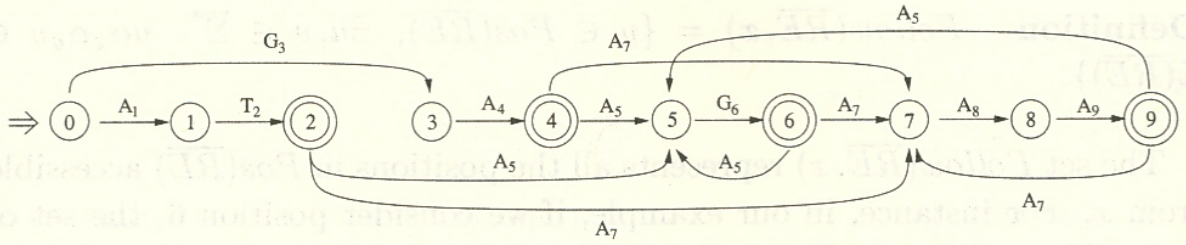


Fig. 5.6. Marked Glushkov automaton built on the marked regular expression $(A_1 T_2|G_3 A_4)((A_5 G_6|A_7 A_8 A_9)*)$. The state 0 is initial. Double-circled states are final.

To obtain the Glushkov automaton of the original RE, we simply erase the position indices in the marked automaton. At this step, the automaton usually becomes nondeterministic. The new automaton recognizes the language $L(RE)$. The Glushkov automaton of our example `(AT|GA)((AG|AAA)*)` is shown in Figure 5.7.

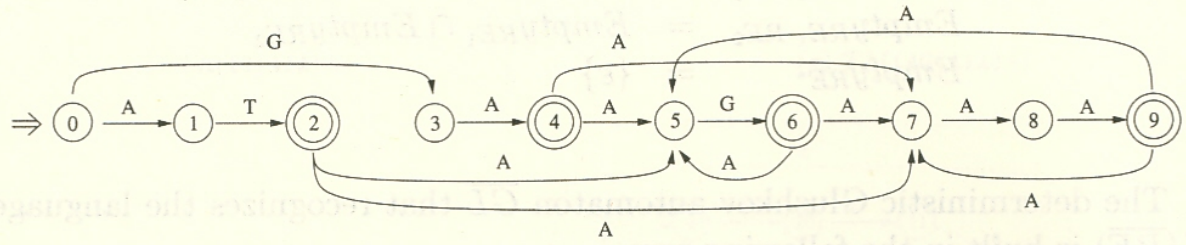


Fig. 5.7. Glushkov automaton built on the regular expression `(AT|GA)((AG|AAA)*)`. The state 0 is initial. Double-circled states are final. The automaton is derived from the marked automaton by simply erasing the position indices.

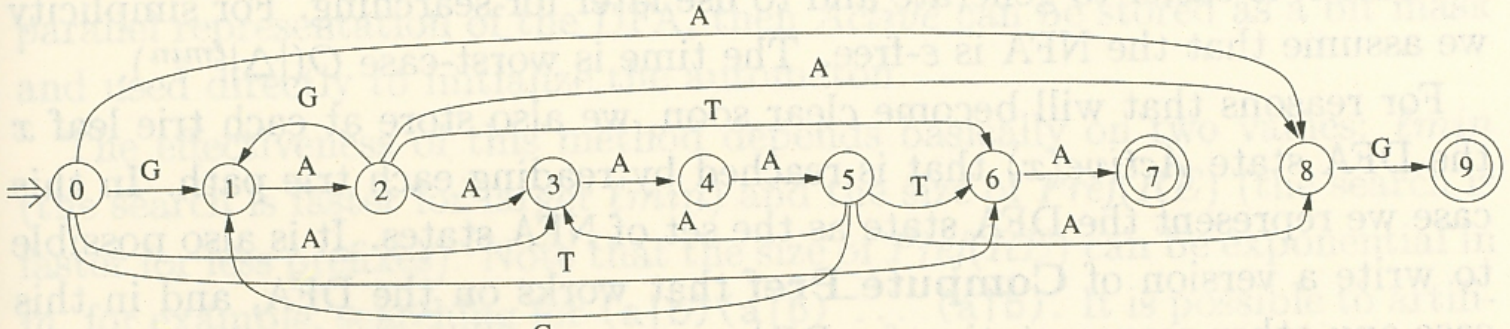
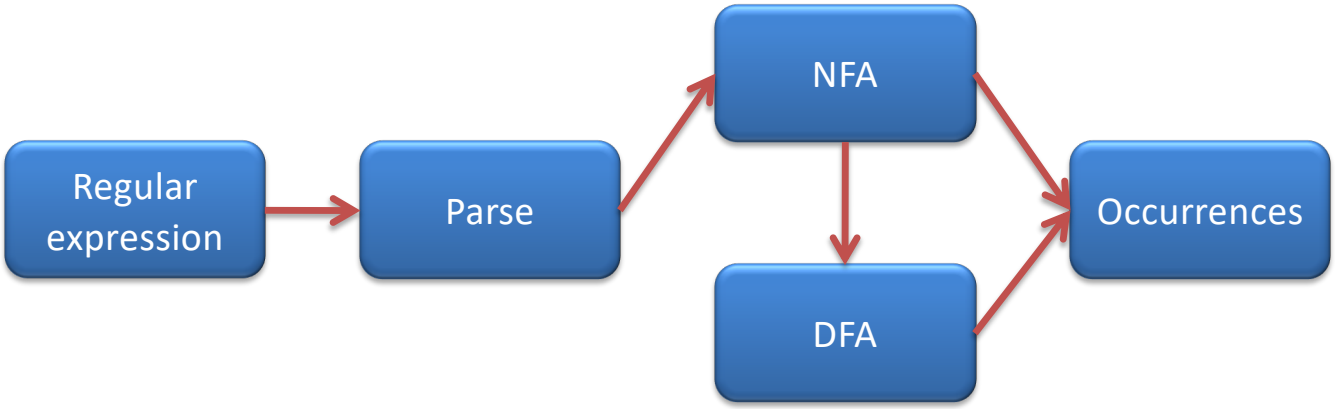


Fig. 5.19. Glushkov automaton built on the regular expression `((GA|AAA)*)(TA|AG)`.

- No ε links
- All incoming arcs have the same character label
- To reach a certain state always the same character from text had to be read.
- Construction: worst case is $O(m^3)$ since poor performance for star closures...
- But this has been speeded up a bit

Matching of RE-s



NFA -> DFA

- Why?
- More straightforward (i.e. faster) to match/simulate

Determinization of a NFA into a DFA

- Maintain at each stage a set of states reachable from previous set on the given character. (Remove ε transitions.)
- Represent every reachable combination of states of a NFA as a new state of DFA
- From each state there has to be only one transition on a given character.
- Automata for Matching Patterns Handbook of Formal Languages (kohalik)
- **Navarro and Raffinot** Flexible Pattern Matching in Strings. (Cambridge University Press, 2002). pp. 106 -> pp 115
-

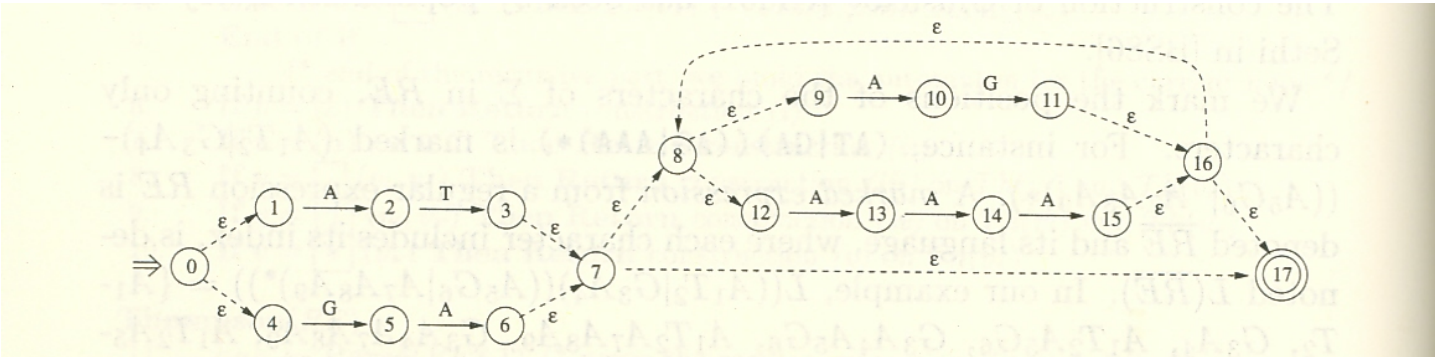


Fig. 5.5. Thompson automaton construction for the regular expression `(AA|AT)((AG|AAA)*)`.

Maintain at each stage **a set of states reachable** from previous set **on the given character**. (Remove ε transitions.)

Represent every reachable **combination of states of a NFA as a new state of DFA**

From each state there can be **only one transition on a given character.**

```
BuildState(S)
1.    If S ∩ F ≠ ∅ Then F_d ← F_d ∪ {S}
2.    For σ ∈ Σ Do
3.        T ← ∅
4.        For s ∈ S Do
5.            For (s, σ, s') ∈ Δ Do T ← T ∪ E(s')
6.        End of for
7.        δ(S, σ) ← T
8.        If T ∉ Q_d Then
9.            Q_d ← Q_d ∪ {T}
10.           BuildState(T)
11.       End of if
12.   End of for

BuildDFA(N = (Q, Σ, I, F, Δ))
13.   EpsClosure(N)
14.   I_d ← E(I) /* initial DFA state */
15.   F_d ← ∅ /* final DFA states */
16.   Q_d ← {I_d} /* all the DFA states */
17.   BuildState(I_d)
18.   Return (Q_d, Σ, I_d, F_d, δ)
```

Fig. 5.11. Classical computation of the DFA from the NFA.

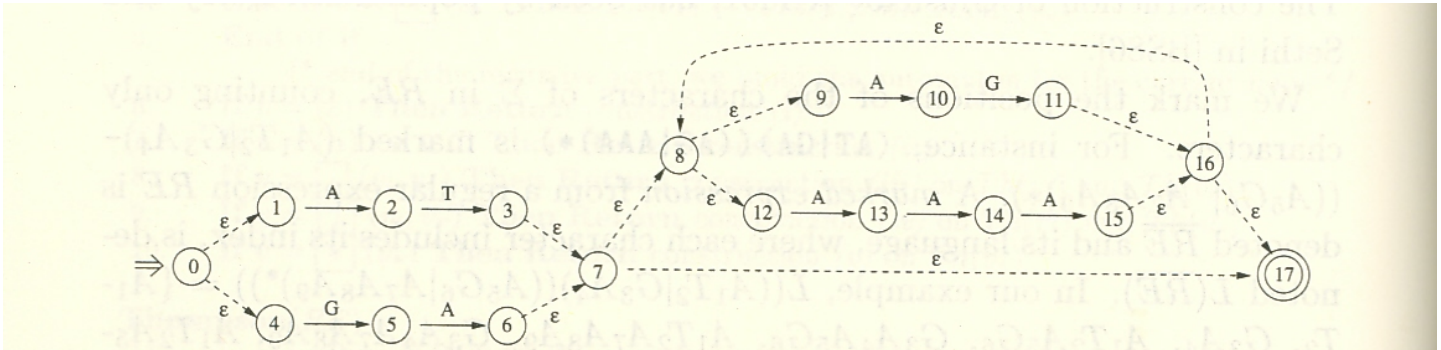


Fig. 5.5. Thompson automaton construction for the regular expression `(AA|AT)((AG|AAA)*)`.

	States	A	T
E(0)	0,1,4	2	-
E(1)	2	-	3,7,8,9,12,17
E(2)	3,7,8,9,12,17		

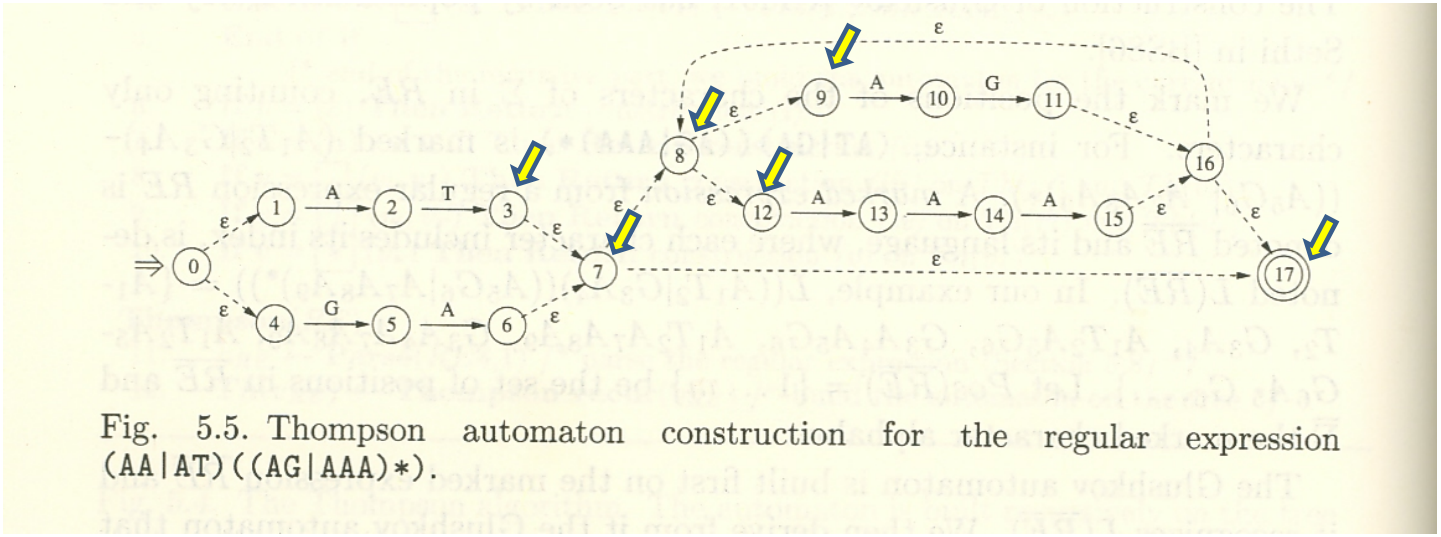


Fig. 5.5. Thompson automaton construction for the regular expression `(AA|AT)((AG|AAA)*)`.

	States	A	T
E(0)	0,1,4	2	-
E(1)	2	-	3,7,8,9,12,17
E(2) F	3,7,8,9,12,17		

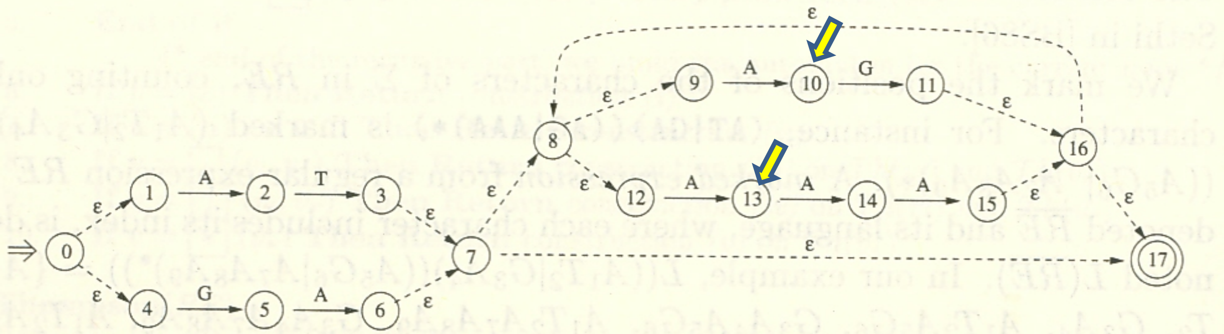


Fig. 5.5. Thompson automaton construction for the regular expression (AA|AT)((AG|AAA)*).

	States	A	T
E(0)	0,1,4	2	-
E(1)	2	-	3,7,8,9,12,17
E(2) F	3,7,8,9,12,17	10,13	-

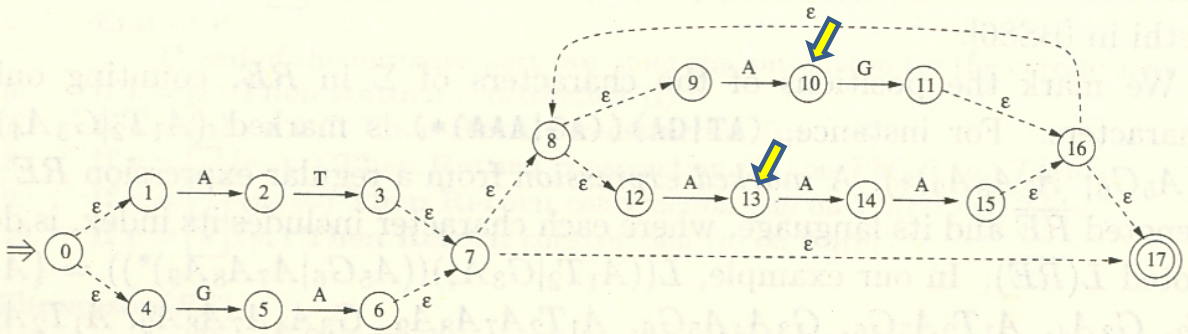


Fig. 5.5. Thompson automaton construction for the regular expression `(AA|AT)((AG|AAA)*)`.

	States	A	T
E(0)	0,1,4	2	-
E(1)	2	-	3,7,8,9,12,17
E(2) F	3,7,8,9,12,17	10,13	-
E(3)	10,13		

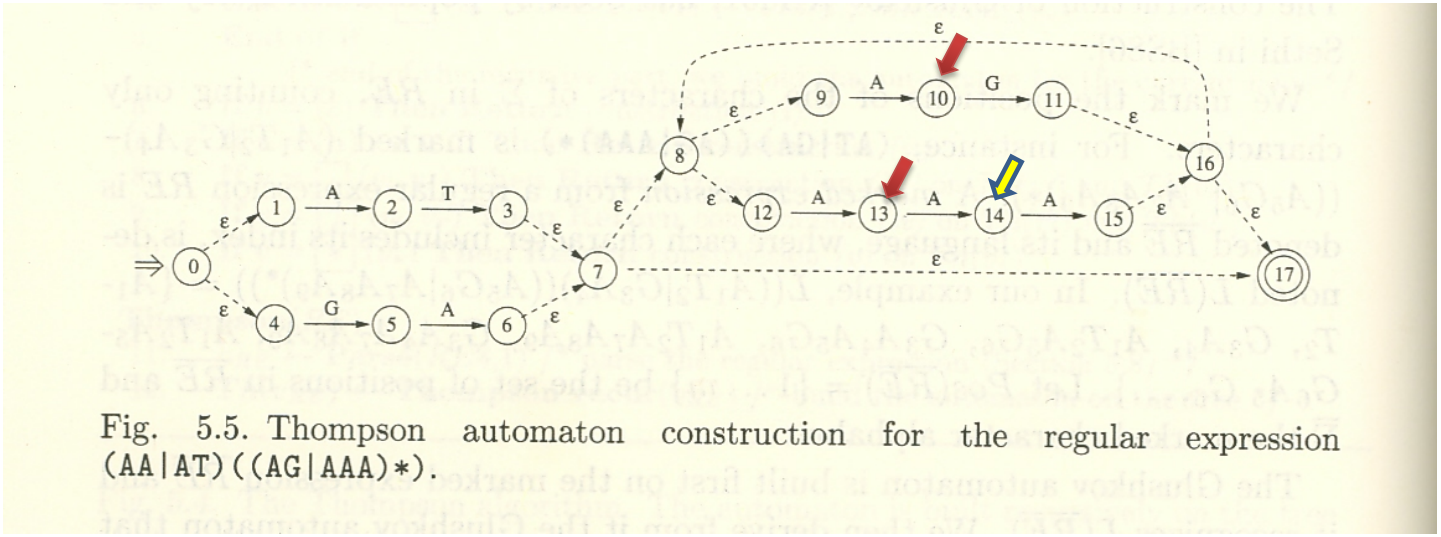


Fig. 5.5. Thompson automaton construction for the regular expression `(AA|AT)((AG|AAA)*)`.

	States	A	T	G
E(0)	0,1,4	2	-	-
E(1)	2	-	3,7,8,9,12,17	
E(2) F	3,7,8,9,12,17	10,13	-	-
E(3)	10,13	14	-	

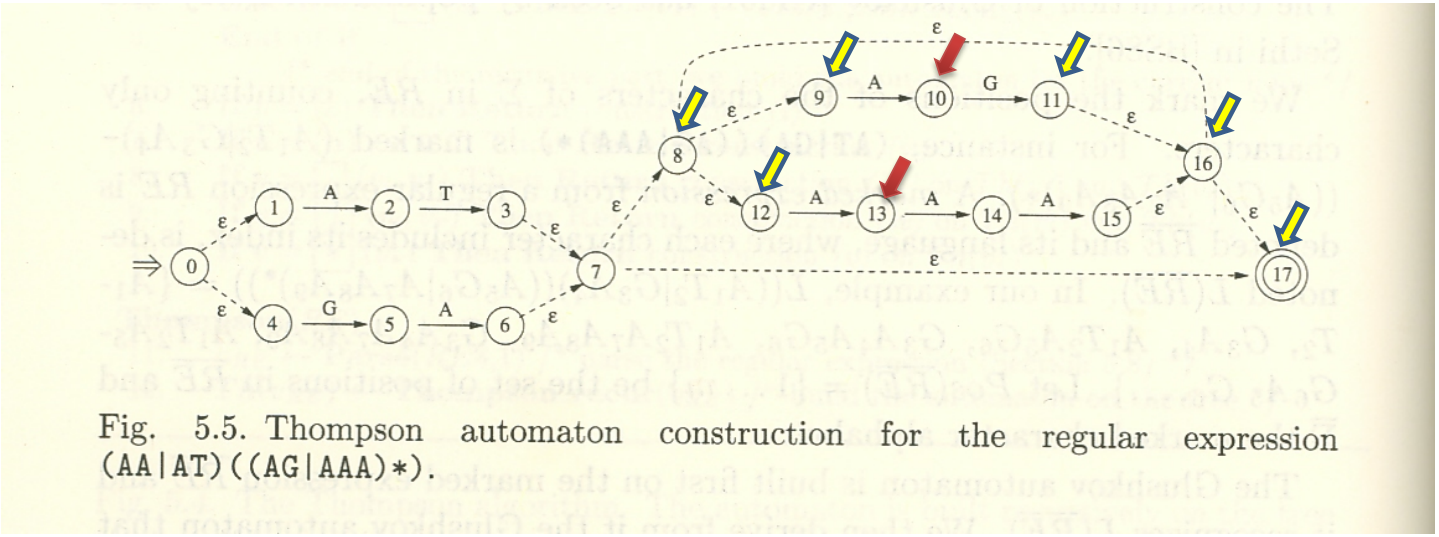


Fig. 5.5. Thompson automaton construction for the regular expression `(AA|AT)((AG|AAA)*)`.

	States	A	T	G
E(0)	0,1,4	2	-	- (5?)
E(1)	2	-	3,7,8,9,12,17	
E(2) F	3,7,8,9,12,17	10,13	-	-
E(3)	10,13	14	-	11,16,17,8,9,12

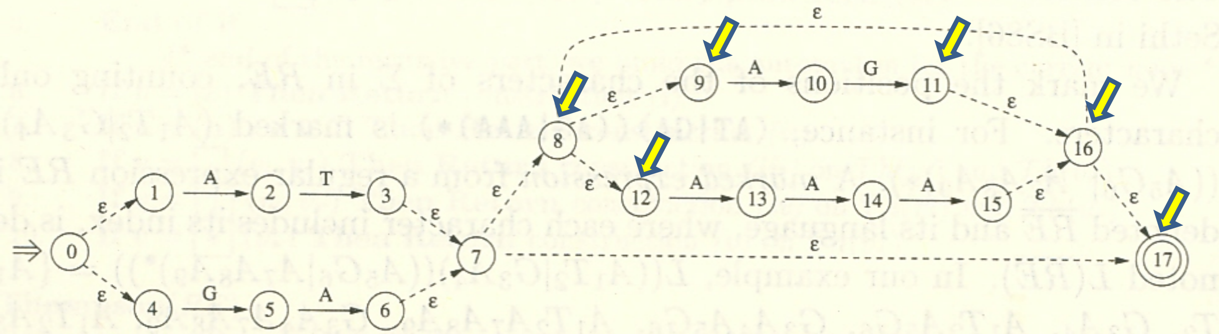


Fig. 5.5. Thompson automaton construction for the regular expression (AA|AT)((AG|AAA)*).

DFA state	NFA States	A	T	G
0	0,1,4	2	-	-
1	2	-	3,7,8,9,12,17	
2 F	3,7,8,9,12,17	10,13	-	-
3	10,13	14	-	8,9,11, 12, 16,17
4	14	15,16,17,8,9,12	-	
5	8,9,12,15,16,17	10,13	-	-
6	8,9,11,12,16,17	10,13	-	-
7	10,13	14	-	11,16,17,8,9,12

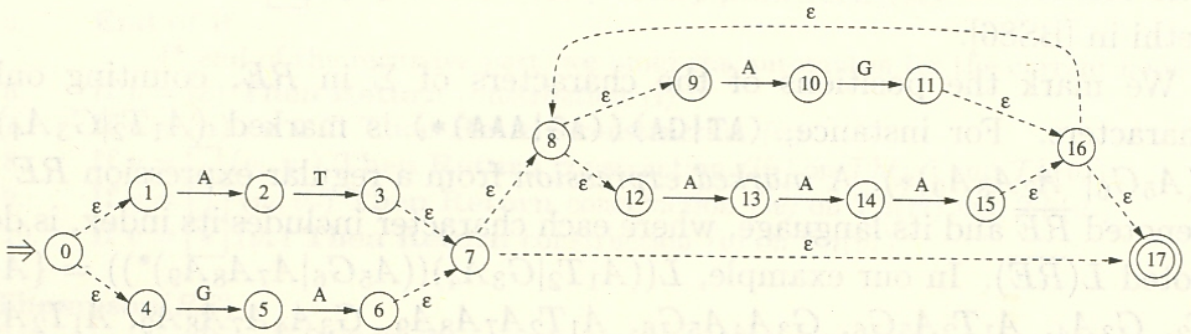


Fig. 5.5. Thompson automaton construction for the regular expression `(AA|AT)((AG|AAA)*)`.

DFA state	NFA States	A	T	G
0	0,1,4	2	-	5
1	2	-	3,7,8,9,12,17	
2 F	3,7,8,9,12,17	10,13	-	-
3	10,13	14	-	8,9,11, 12, 16,17
4	14	8,9,12,15,16,17	-	-
5	8,9,11,12,16,17	10,13	-	-
6	8,9,12,15,16,17	10,13	-	-
7	5	6,7,8,9,12,17	-	-
8	6,7,8,9,12,17	10,13	-	-

DFA state	NFA States	A	T	G
0	0,1,4	2	-	5
1	2	-	3,7,8,9,12,17	
2 F	3,7,8,9,12,17	10,13	-	-
3	10,13	14	-	8,9,11, 12, 16,17
4	14	8,9,12,15,16,17	-	-
5	8,9,11,12,16,17	10,13	-	-
6	8,9,12,15,16,17	10,13	-	-
7	5	6,7,8,9,12,17	-	-
8	6,7,8,9,12,17	10,13	-	-

State	A	T	G
0	1	-	7
1	-	2	
2 F	3	-	-
3	4	-	6
4	5	-	-
5 F	3	-	-
6 F	3	-	-
7	8	-	-
8 F	4	-	-

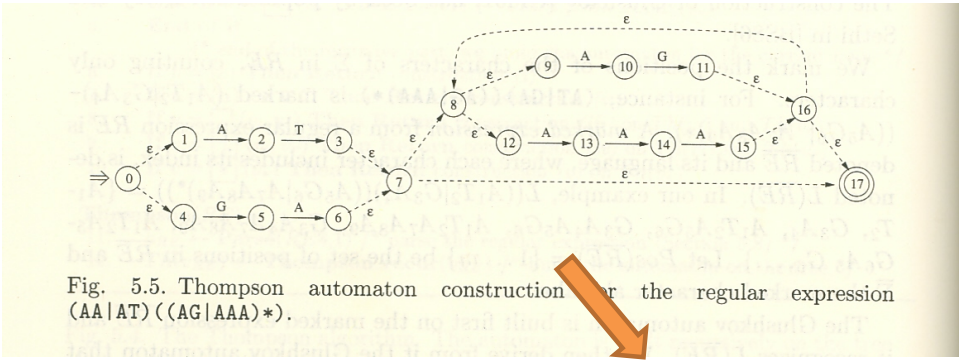
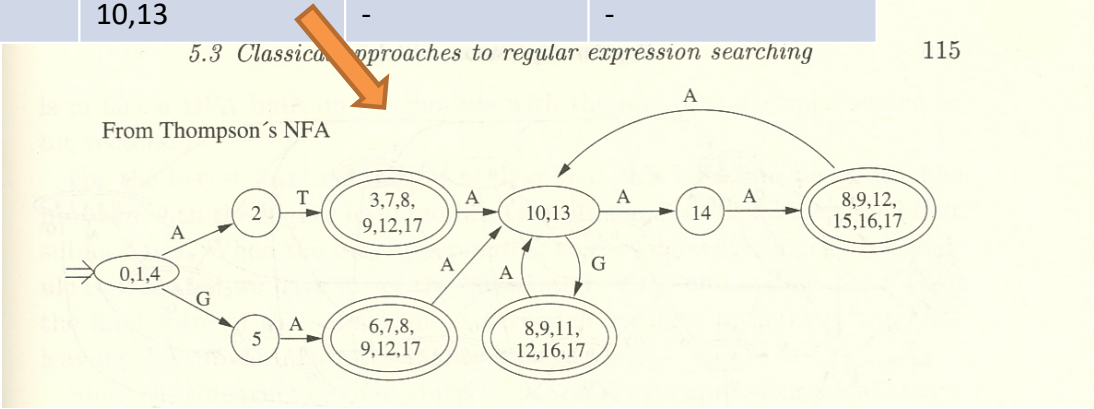


Fig. 5.5. Thompson automaton construction for the regular expression `(AA|AT)((AG|AAA)*)`.

DFA state	NFA States	A	T	G
0	0,1,4	2	-	5
1	2	-	3,7,8,9,12,17	
2 F	3,7,8,9,12,17	10,13	-	-
3	10,13	14	-	8,9,11, 12, 16,17
4	14	8,9,12,15,16,17	-	-
5	8,9,11,12,16,17	10,13	-	-
6	8,9,12,15,16,17	10,13	-	-
7	5	6,7,8,9,12,17	-	-
8	6,7,8,9,12,17	10,13	-	-



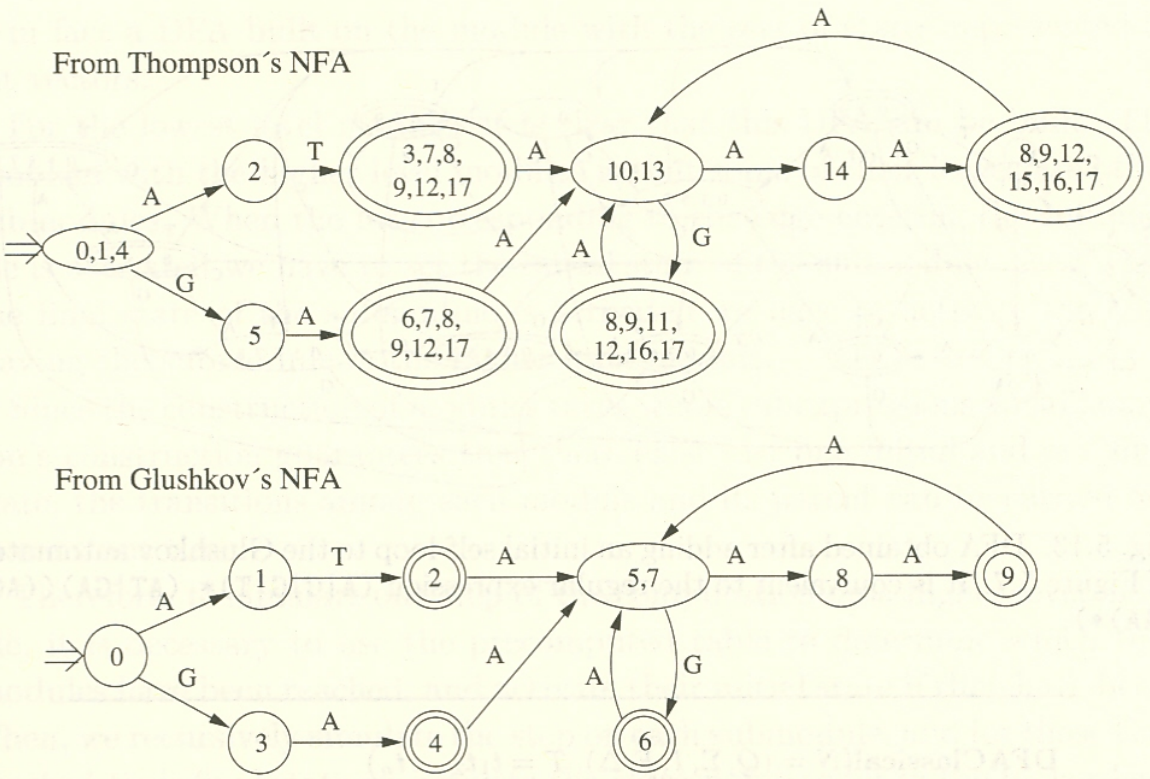


Fig. 5.12. The DFAs resulting from Thompson's and Glushkov's NFAs.

Minimization of automata

- DFA construction does not always produce the minimal automaton
- Smaller -> better(?)
- Must still represent equivalent languages!

Minimization of automata

- Language is simply a subset of all possible strings of Σ*
- Regular language is the language that can be described using regular expressions and recognized by a finite automaton (no pushdown, multi-tape, or Turing machines)
- Two automatons that recognize exactly the same language are **equivalent**
- The automaton that has fewest possible states from the same equivalence class, is the **minimal**
- Automaton with more states is called **redundant**
- The automaton construction techniques do not create the minimal automatons
- It would be easier to understand nonredundant automatons
- Smaller automaton consumes less memory
- The manipulation is faster
-

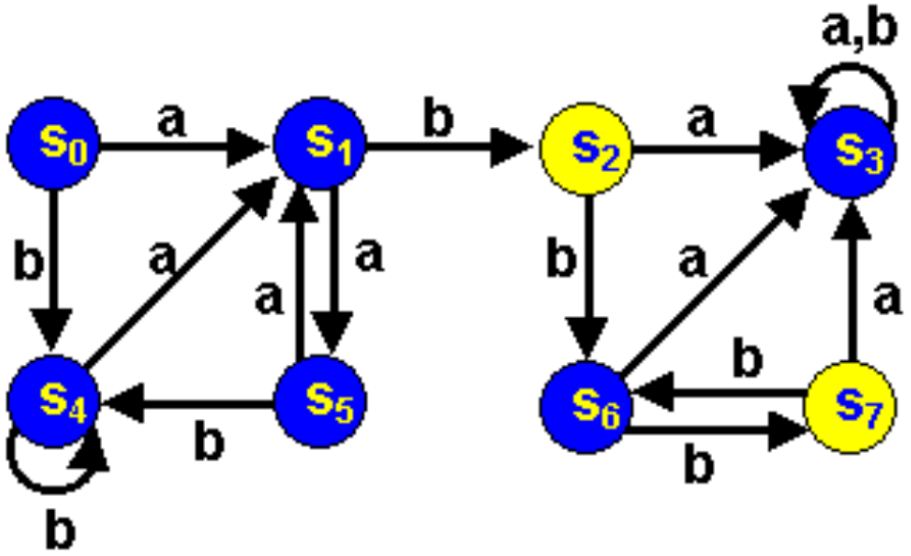
Minimization

- A compiler course subject
- Aho, Sethi, Ullmann Compilers -- Principles, Techniques and Tools. Addison Wesley 1986. (The Dragon Book)
- Minimization description (L4_RegExp/min-fa.html)

  A: Merge all equivalent states until minimum achieved

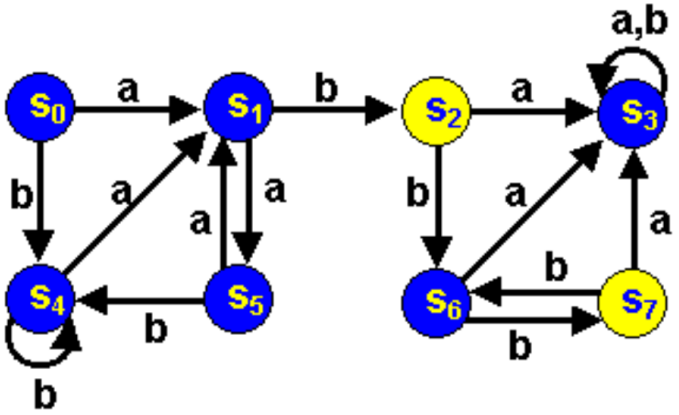
  B: Start from minimal possible (2-state) and split states until no conflicts

- Fact. Equivalent states go to equivalent states under all inputs.
- Recognizer for (aa | b)*ab(bb)*



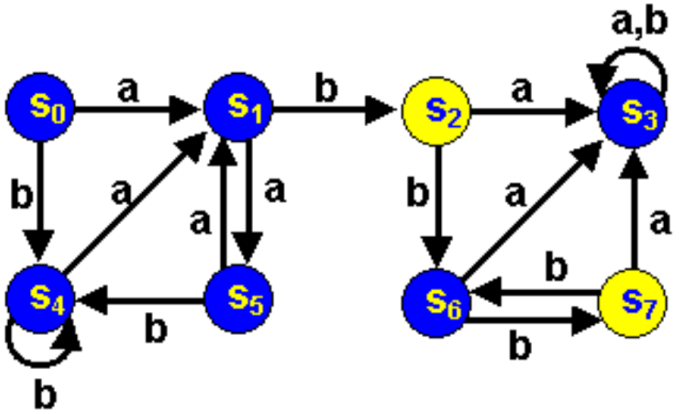
Step 1: Generate 2 equivalence classes: final and other states

	a	b	
2	3:B	6:B	A
7	3:B	6:B	
---	---	---	
0	1:B	4:B	
1	5:B	2:**A**	
3	3:B	3:B	
4	1:B	4:B	B
5	1:B	4:B	
6	3:B	7:**A**	



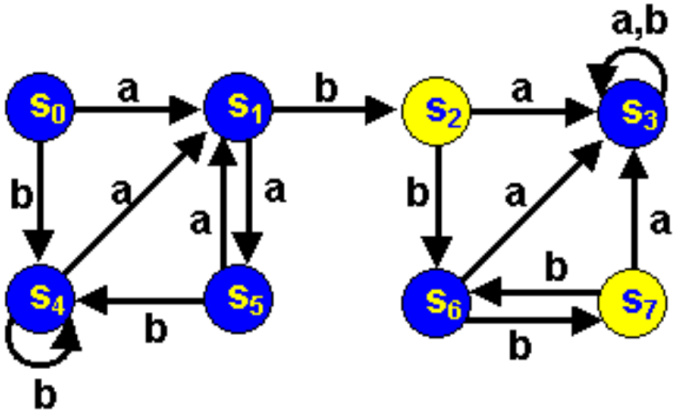
Step 2: Create new class from 1 and 6 (conflict on b)

	a	b	
2	3:B	6:C	A
7	3:B	6:C	
---	---	---	---
0	1:C	4:B	
3	3:B	3:B	B
4	1:C	4:B	
5	1:C	4:B	
---	---	---	---
1	5:B	2:A	C
6	3:C	7:A	



Step 3: Create new class from 3

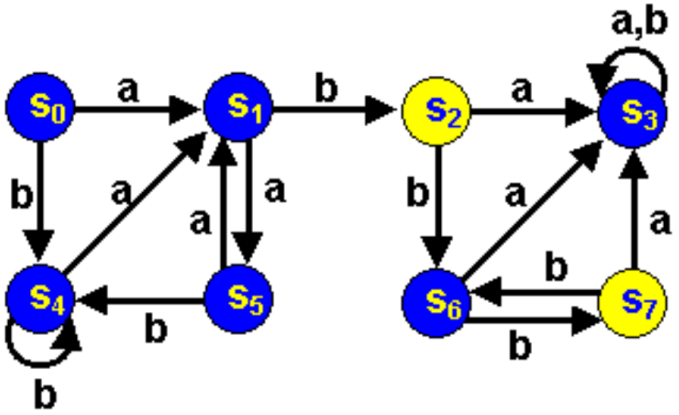
	a	b	
2	3:D	6:C	A
7	3:D	6:C	
---	---	---	---
0	1:C	4:B	
4	1:C	4:B	B
5	1:C	4:B	
---	---	---	---
1	5:B	2:A	C
6	3:**D**	7:A	
---	---	---	---
3	3:D	3:D	D



Step 4: Create new class from 6

	a	b	
2	3:D	6:E	A
7	3:D	6:E	
---	---	---	---
0	1:C	4:B	
4	1:C	4:B	B
5	1:C	4:B	
---	---	---	---
1	5:B	2:A	C
---	---	---	---
3	3:D	3:D	D
---	---	---	---
6	3:D	7:A	E

All states are consistent



Minimal automaton

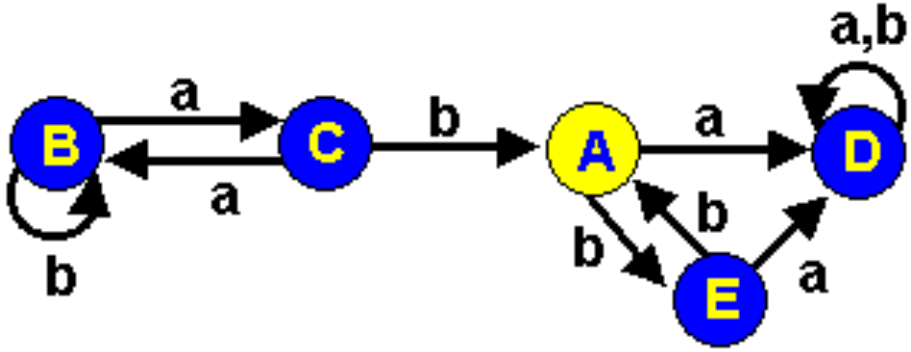
Step 4: Create new class from 6

	a	b	
2	3:D	6:E	A
7	3:D	6:E	
---	---	---	
0	1:C	4:B	
4	1:C	4:B	B
5	1:C	4:B	
---	---	---	
1	5:B	2:A	C
---	---	---	
3	3:D	3:D	D
---	---	---	
6	3:D	7:A	E

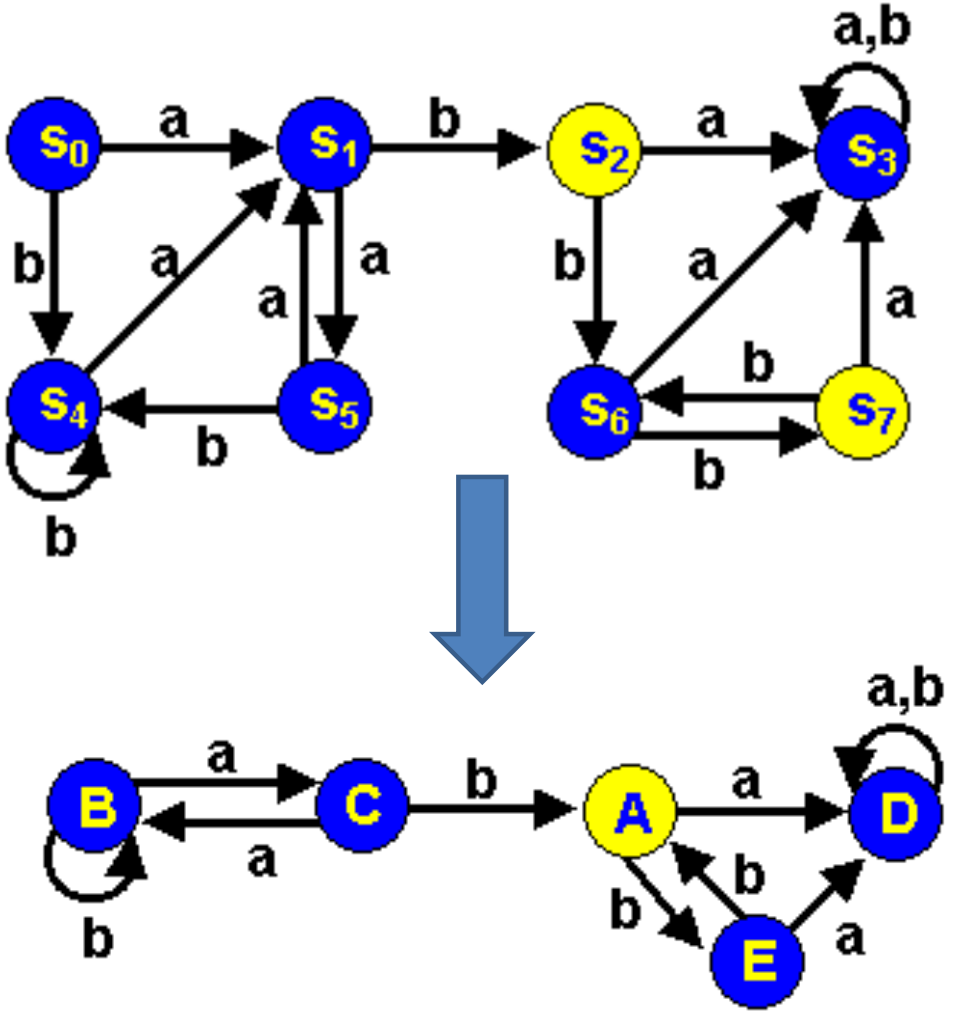
All states are consistent

	a	b
A	D	E
B	C	B
C	B	A
D	D	D
E	D	A

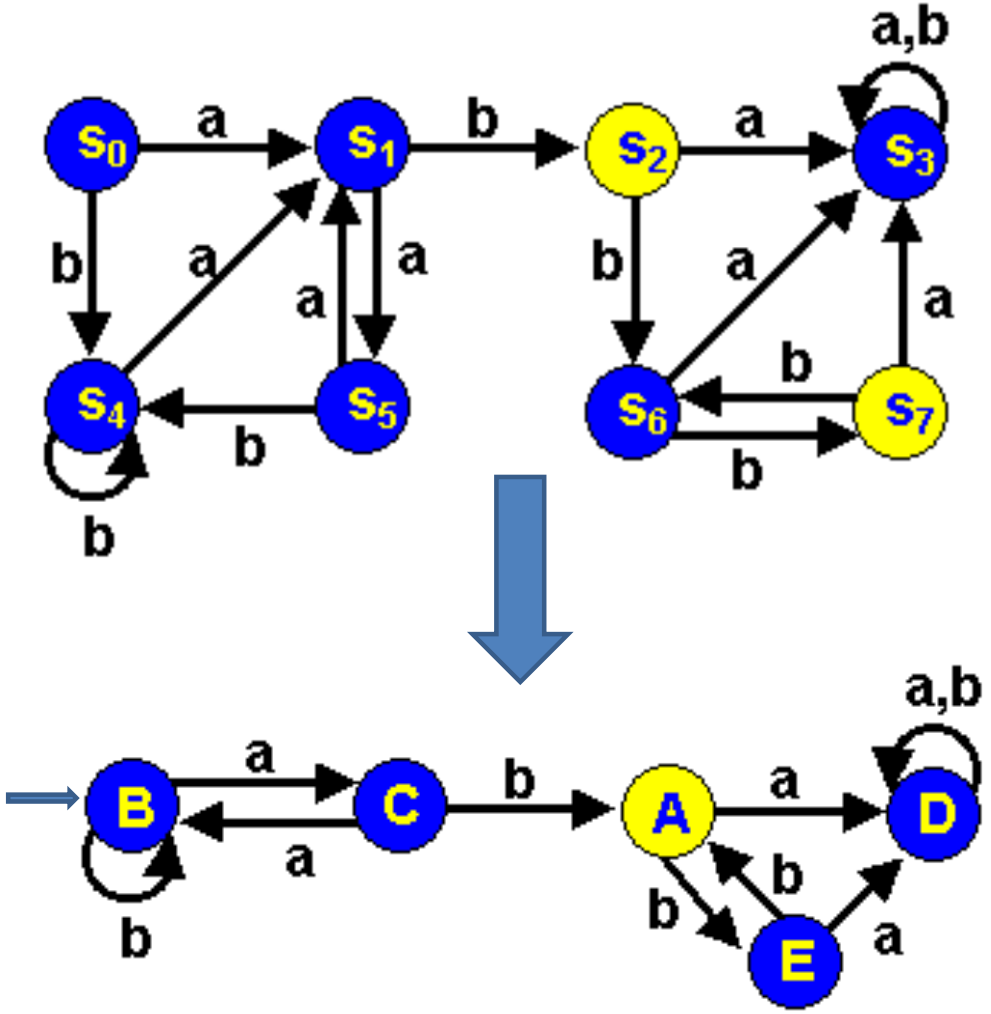
	a	b
A	D	E
B	C	B
C	B	A
D	D	D
E	D	A



(aa | b)*ab(bb)*



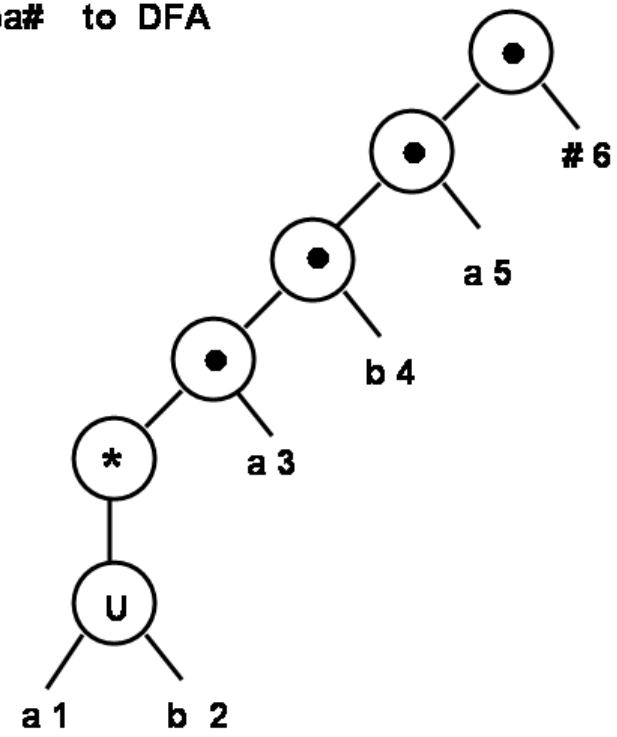
(aa | b)*ab(bb)*



Construct a DFA from the regular expression

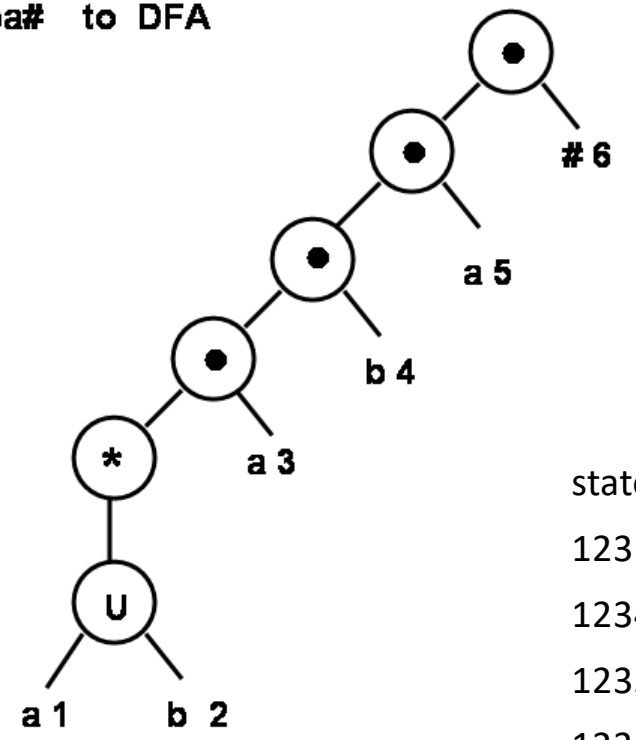
- Usually NFA is constructed, and then determinized
- McNaughton and Yamada proposed a method for direct construction of a DFA
- R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. IEEE Transactions on Electronic Computers, 9(1):39--47, March 1960.

(a | b)* aba# to DFA



- Example: Lets analyze RE = (a ∪ b)*aba
- Add end symbol # : (a ∪ b)*aba#
- Make a parse tree
  - Leaves represent symbols of Σ from RE
  - Internal nodes - operators
- Give a unique numbering of leaves
- Position nr is **active** if this **can represent the next symbol**
- DFA states and transitions are made from the tree:
- A state of DFA corresponds to a set of positions that are active after reading some prefix of the input
- Initial state is (1,2,3) (when nothing has been read yet)
- DFA contains transuitions $q \rightarrow_a q'$, whwre q' are position nrs that are activated when in positions of q the character a is read.
- Final states are those containing the position number of #

(a | b)* aba# to DFA

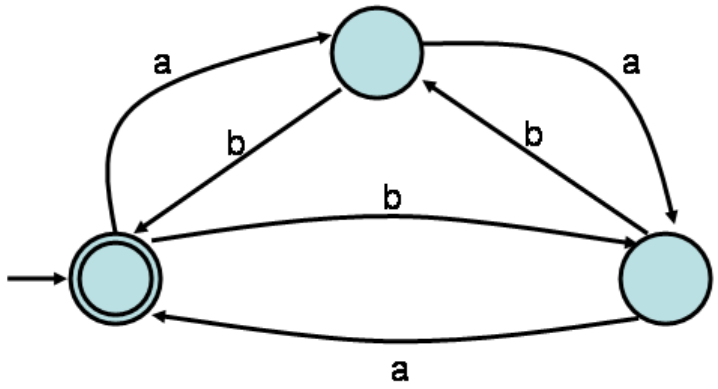


state	a	b
123 (start)	1234	123
1234	1234	1235
1235	12346	123
12346	1234	1235

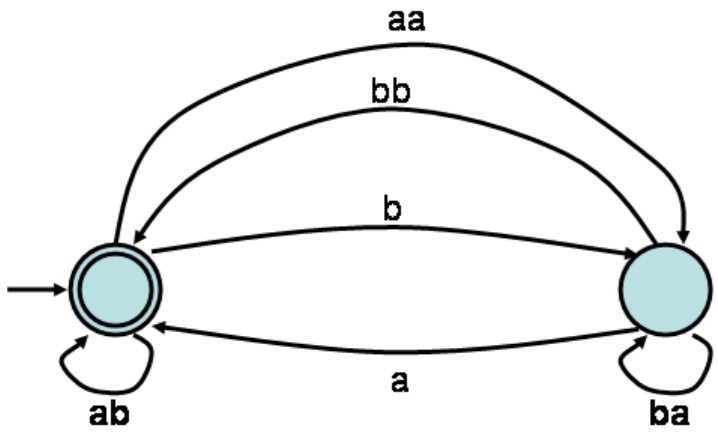
Construction of regular expressions from the automata

- It is possible to start from an automaton and then generate the regular expression that describes the language recognized by the automaton.
- Example. http://biit.cs.ut.ee/~vilo/edu/2002-03/Tekstialgoritmid_I/Praksid/Lahendused_3/3/3.html
- The program JFLAP for transforming FSA to regular expressions can be downloaded from http://www.jflap.org/, or http://www.cs.duke.edu/~rodger/tools/jflap/indexold.html
- In the bottom of page there are links to "current version".

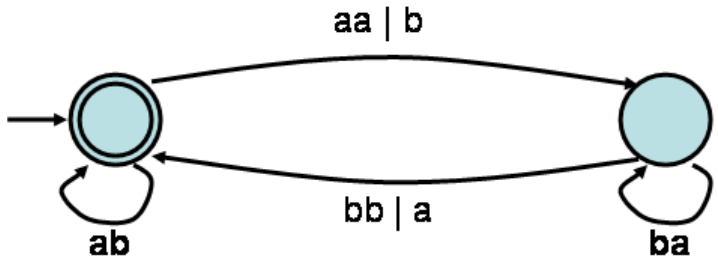
Moodusta reg. avaldis sellest automaadist



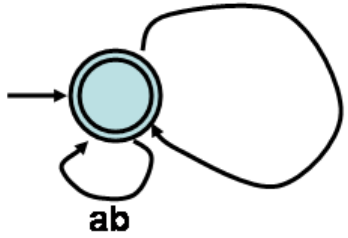
Moodusta reg. avaldis sellest automaadist



Moodusta reg. avaldis sellest automaadist

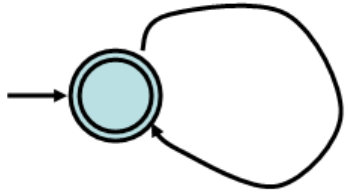


Moodusta reg. avaldis sellest automaadist



(aa | b) (ba)* (bb | a)

Moodusta reg. avaldis sellest automaadist



ab | (aa | b) (ba)* (bb | a)

Moodusta reg. avaldis sellest automaadist

(ab | (aa | b) (ba)* (bb | a))*

Filtering approaches for regular expression searches

- Identify a (sub)set of prefixes or factors that are necessarily present in the language represented by regexp.
- Use multi-pattern matching techniques for matching them all simultaneously
- In case of a match use the automaton to verify the occurrence

- Prefixes
- lmin - the shortest occurrence length (to avoid missing short occurrences)
- ((GA|AAA)*(TA|AG)) the set of 2-long prefixes is { GA, AA, TA, AG }
- (AT|GA)(AG|AAA)((AG|AAA)+) lmin=6
- { ATAGAG, ATAGAA, ATAAAA, GAAGAG, GAAGAA, GAAAAA }

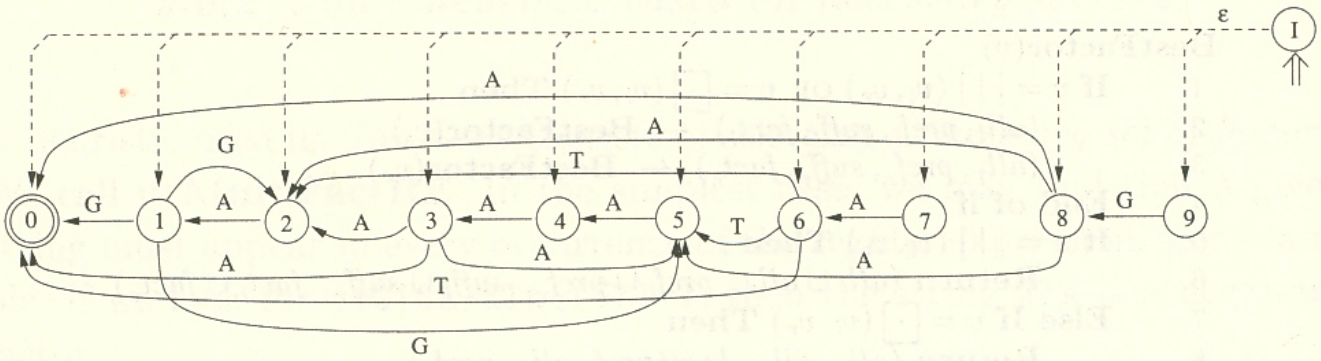
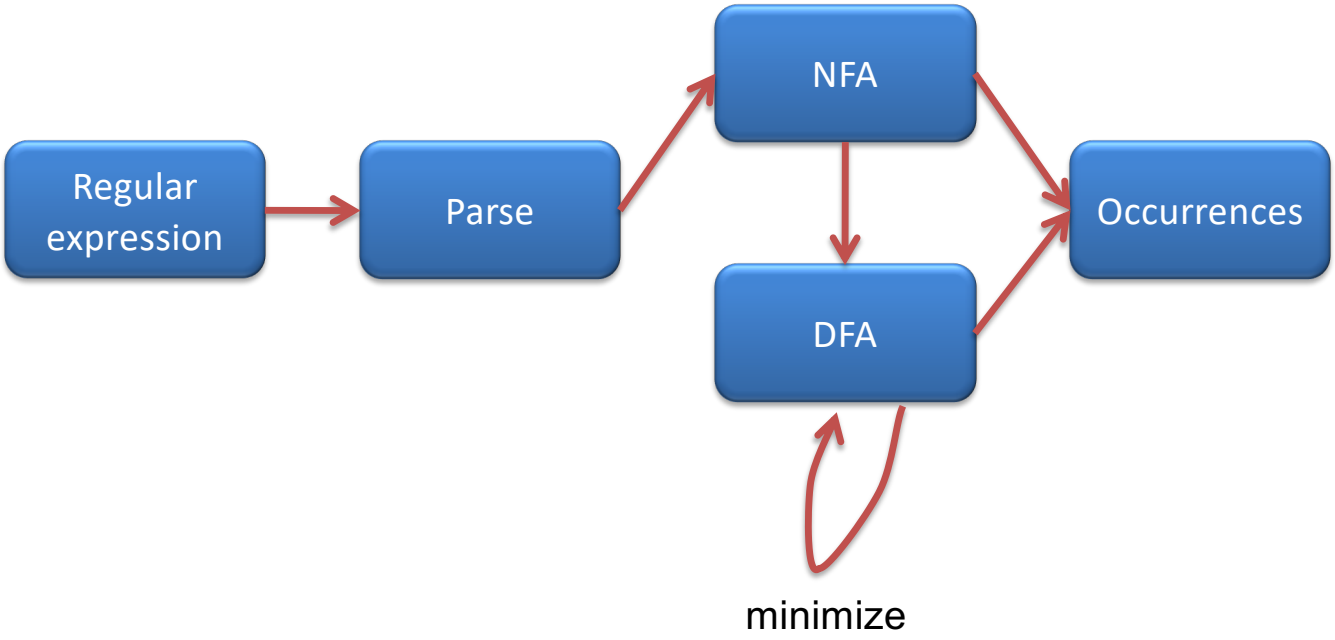


Fig. 5.24. Automaton to recognize all the reverse prefixes of the regular expression `((GA|AAA)*)(TA|AG)`.

- (AG|GA)ATA((TT)*)
- The string ATA is a necessary factor.
- Gnu grep uses such heuristics
- Can be developed to utilise a lot of knowledge about possible frequences of occurrences, speed of multi-pattern matchers etc.

Summary



How to find primes

```
$ perl -e 'foreach $i ( 1000..1050 ) { print "$i\n" if ("a"x$i) !~ /^(aa+)(\1+)$/ }'
1009
1013
1019
1021
1031
1033
1039
1049
$
```

Learning languages

- AGAGGAT +
- ATGAGAA +
- ATGATTA –
- AA –
- AAATGA –
- AGATAG +

Grammatical inference

Q: What is the language represented by the positive examples?

A1: List of positive examples

A2: Minimal automaton that recognises + examples, and none of the – examples?

Finding A2 in general a computationally hard problem